Self-stabilizing minimum-degree spanning tree within one from the optimal degree

Lélia Blin, Maria Gradinariu Potop-Butucaru, Stephane Rovedakis

Self-stabilizing minimum-degree spanning tree within one from the optimal degree

Lélia Blin[1,2] Maria Gradinariu Potop-Butucaru[2,3] Stéphane Rovedakis[1]

Abstract

We propose a self-stabilizing algorithm for constructing a Minimum-Degree Spanning Tree (**MDST**) in undirected networks. Starting from an arbitrary state, our algorithm is guaranteed to converge to a legitimate state describing a spanning tree whose maximum node degree is at most $\Delta^* + 1$, where Δ^* is the minimum possible maximum degree of a spanning tree of the network.

To the best of our knowledge our algorithm is the first self-stabilizing solution for the construction of a minimum-degree spanning tree in undirected graphs. The algorithm uses only local communications (nodes interact only with the neighbors at one hop distance). Moreover, the algorithm is designed to work in any asynchronous message passing network with reliable FIFO channels. Additionally, we use a fine grained atomicity model (i.e. the send/receive atomicity). The time complexity of our solution is $O(mn^2 \log n)$ where m is the number of edges and n is the number of nodes. The memory complexity is $O(\delta \log n)$ in the send-receive atomicity model (δ is the maximal degree of the network).

1 Introduction

The spanning tree construction is a fundamental problem in the field of distributed network algorithms being the building block for a broad class of fundamental services: communication primitives (e.g. broadcast or multicast), deadlock resolution or mutual exclusion. The new emergent distributed systems such as ad-hoc networks, sensor or peer-to-peer networks zoom on the cost efficiency of distributed communication structures and in particular spanning trees. The main issue in adhoc network systems for example is the communication cost. If the communication overlay contains nodes with large degree then undesirable effects can be observed: congestion, collisions or traffic burst. Moreover, these nodes are also the first targets in security attacks. In such case, the construction of reliable spanning trees in which the degree of a node is the lowest possible is needed.

Interestingly, in peer-to-peer networks the motivation for the construction of minimum degree trees is motivated by the nodes (users) welfare. Each communication on behalf of

[1]Université d'Evry, IBISC, CNRS, France.
[2]Univ. Pierre & Marie Curie - Paris 6, LIP6-CNRS UMR 7606, France.
[3]INRIA REGAL, France.

other nodes in the network minimizes the possibility of a node to use the bandwidth for its own interests. As immediate consequence: nodes are incited to cheat on their real bandwidth or to have a free ridding behavior and illegally exploit the local traffic.

Self-stabilization as sub-field of fault-tolerance was first introduced in distributed computing area by Dijkstra in 1974 [6, 18]. A distributed algorithm is self-stabilizing if, starting from an arbitrary state, it guarantees to converge to a legitimate state in finite number of steps and to remain in a legitimate set of states thereafter. The property of self-stabilization enables a distributed algorithm to recover from a transient fault regardless of its initial state.

A broad class of self-stabilizing spanning trees have been proposed so far: by example [1, 7, 2] propose BFS trees, [5, 4] compute minimum diameter spanning trees, and [12] calculate minimum weight spanning trees. A detailed survey on the different self-stabilizing spanning tree schemes can be found in [18, 11]. Recently, [13] propose solutions for the construction of constant degree spanning trees in dynamic settings. To our knowledge there is no self-stabilizing algorithm for constructing minimum degree spanning trees.

This paper tackles the self-stabilizing construction of minimum-degree spanning tree in undirected graphs. More precisely, let $G = (V, E)$ be a graph. Our objective is to compute a spanning tree of G that has minimum degree among all spanning trees of G, where the degree of the tree is the maximum degree of its nodes. In fact, since this problem is NP-hard (by reduction from the Hamiltonian path problem), we are interested in constructing a spanning tree whose degree is within one from the optimal. This bound is achievable in polynomial-time as proved by Fürer and Raghavachari [8, 9] who describe an incremental sequential algorithm that constructs a spanning tree of degree at most $\Delta^* + 1$, where Δ^* is the degree of a MDST. Blin and Butelle proposed in [3] a distributed version of the algorithm by Fürer and Raghavachari. The algorithm in [3] uses techniques similar to those in [10] for controlling and updating fragment sub-trees of the currently constructed spanning trees. None of the previously mentioned solutions are self-stabilizing.

Our results We describe a self-stabilizing algorithm for constructing a minimum degree spanning tree in arbitrary networks. Our contribution is twofold. First, to the best of our knowledge our algorithm is the first self-stabilizing approximate solution for the construction of a minimum-degree spanning tree in undirected graphs. The algorithm converges to a legitimate state describing a spanning tree whose maximum node degree is at most $\Delta^* + 1$, where Δ^* is the minimum possible degree of a spanning tree of the network. Note that computing Δ^* is NP-hard. The algorithm uses only local communications — nodes interact only with their one hop neighbors — which makes it suitable for large scale and scalable systems. Our algorithm is designed to work in any asynchronous message passing network with reliable FIFO channels. Additionally, we use a fine grained atomicity model (i.e. *send/receive atomicity*) defined for the first time in [5][1].

Secondly, our approach is based on the detection of fundamental cycles (i.e. cycles obtained by adding one edge to a tree) contrary to the technique used in [3] that perpetually updates membership information with respect to the different fragments (sub-trees)

[1]The send/receive atomicity is the message passing counter-part of the read/write atomicity defined for the register model [18]

co-existing in the network. As a consequence, and in contrast with [3], our algorithm is able to decrease simultaneously the degree of each node of maximum degree. The time complexity of our solution is $O(mn^2 \log n)$ where m is the number of edges of the network and n the number of nodes. The memory complexity is $O(\log n)$ in a classical message passing model and $O(\delta \log n)$ in the send/received atomicity model (δ is the maximal degree of the network). The maximal length of messages used by our algorithm is $O(n \log n)$.

Paper road-map The paper is organized as follows. The next section introduces the model and the problem description. Section 3 presents the detailed description of our algorithm, then Section 4 and 5 contain the performance and complexity algorithm analysis respectively (only an abstract of correctness proof is given due to space constraints). The last section of the paper resumes the main results and outlines some open problems.

2 Model and notations

System model We borrow the model proposed in [5]. We consider an undirected connected network $G = (V, E)$ where V is the set of nodes and E is the set of edges. Nodes represent processors and edges represent communication links. Each node in the network has an unique identifier. For a node $v \in V$, we denote the set of its neighbors $\mathcal{N}(v) = \{u, \{u, v\} \in E\}$ ($\{u, v\}$ denotes the edge between the node u and its neighbor v). The size of the set $\mathcal{N}(v)$ is the degree of v. A spanning tree, T for G is a connected sub-graph of G such that $T = (V, E_T)$ and $E_T \subset E$, $|E_T| = |V| - 1$. The degree of a node u in T is denoted by $\deg_T(u)$, and we denote by $\deg(T)$ the degree of T, i.e., $\deg(T) = \max_v \deg_T(v)$.

We consider a static topology. The communication model is asynchronous message passing with FIFO channels (on each link messages are delivered in the same order as they have been sent). We use a refinement of this model: the send/receive atomicity ([5]). In this model each node maintains a local copy of the variables of its neighbors, refreshed via special messages (denoted in the sequel `InfoMsg`) exchanged periodically by neighboring nodes. A *local state* of a node is the value of the local variables of the node, the copy of the local variables of its neighbors and the state of its program counter. A *configuration* of the system is the cross product of the local states of all nodes in the system plus the content of the communication links. An *atomic step* at node p is an internal computation based on the current value of p's local state and a single communication operation (send/receive) at p. An *execution* of the system is an infinite sequence of configurations, $e = (c_0, c_1, \ldots c_i, \ldots)$, where each configuration c_{i+1} follows from c_i by the execution of a single atomic step.

Faults and self-stabilization In the sequel we consider the system can start in any configuration. That is, the local state of a node can be corrupted. Note that we don't make any assumption on the bound of corrupted nodes. In the worst case all the nodes in the system may start in a corrupted configuration. In order to tackle these faults we use self-stabilization techniques.

Definition 1 (self-stabilization) *Let $\mathcal{L}_{\mathcal{A}}$ be a non-empty* legitimacy predicate *of an algorithm $\mathcal{A}$ with respect to a specification predicate Spec such that every configuration satisfying*

$\mathcal{L}_{\mathcal{A}}$ satisfies Spec. Algorithm $\mathcal{A}$ is self-stabilizing *with respect to Spec iff the following two conditions hold:*
*(i) Every computation of $\mathcal{A}$ starting from a configuration satisfying $\mathcal{L}_{\mathcal{A}}$ preserves $\mathcal{L}_{\mathcal{A}}$ (*closure*).*
*(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_{\mathcal{A}}$ (*convergence*).*

Minimum Degree Spanning Tree (MDST) A legitimate configuration for the MDST is a configuration that outputs an unique spanning tree of minimum degree. Since computing a Δ^* minimum degree spanning tree in a given network is NP-hard we propose in the following a self-stabilizing MDST with maximum node degree at most $\Delta^* + 1$.

3 A self-stabilizing MDST Algorithm

The main ingredients used by our MDST approximation algorithm are: (1) a module maintaining a spanning tree; (2) a module for computing the maximum node-degree of the spanning tree; (3) a module for computing the fundamental cycles and (4) a procedure for reducing the maximum node-degree of the spanning tree based on the fundamental cycles computed by the previous module.

One challenge is to design and run these modules concurrently, in a self-stabilizing manner. Note that the core of our algorithm is the reduction procedure that aims at repetitively reducing the degree of the spanning tree until getting a spanning tree of degree at most $\Delta^* + 1$.

The following section proposes the detailed description of our algorithm. The formal description of the algorithm can be found in Figure 2, with its sub-procedures in Figures 1 and 3[2].

3.1 Notations, Data Structures and Elementary procedures

Notations. Our algorithm makes use of the following notions (introduced first in [9]): *improving edge* and *blocking node* defined as follows. Let G be a network and T a spanning tree of G. Let $e = \{u, v\}$ be an edge of G that is not in T. The cycle C_e generated by adding e to T is called *fundamental*. Swapping e with any other edge of C_e (hence an edge in a spanning tree T) results in another spanning tree T' of G. Let w be a node of C_e, distinct from u and v, that has maximum degree in T among all nodes of C_e (i.e. $deg_T(w) = \deg(T)$). If the exchange between e and an edge in C_e incident to w decreases by one the degree of w, then e is called an *improving edge*. An improving edge e satisfies

$$\deg_T(w) \geq \max(\deg_T(u), \deg_T(v)) + 2. \quad (1)$$

Let k be the degree of T (i.e., $k = \deg(T)$). If $\deg_T(u) = k - 1$ then u is called a *blocking* node for C_e. If u or v are blocking nodes for C_e then Eq. 1 states that e is not an improving edge for T.

[2]In the proposed algorithms $\oplus$ concatenates lists

```
1  Update_State(u, deg_u)
2  use rules 1 and 2 to correct the local state of v
3  edge_status_v[u] ← is_tree_edge(v, u); deg_v[u] ← deg_u;
4  color_tree_v ← degree_stabilized(v);

5  Action_on_Cycle(idblock, y, path, s)
6  if idblock = NIL then
7      let d_path = max{deg_u : u ∈ path}
8      if dmax_v = d_path then
9          if max{deg_v, deg_{v,y}} = dmax_v − 1 then
10             Deblock(y, s);
11         else
12             if max{deg_v, deg_{v,y}} < dmax_v − 1 then
13                 let w = min{ID_u : deg_u =
                   d_path, u ∈ path} and (w, z) ∈ path
14                 Improve(y, deg_w, (w, z), path);
15 else if idblock ∈ path then
16     if max{deg_v, deg_{v,y}} = dmax_v − 1 then
17         Deblock(y, s);
18     else
19         if max{deg_v, deg_{v,y}} < dmax_v − 1 then
20             let e = (idblock, z) ∈ path
21             Improve(y, deg_idblock, e, path);

22 Broadcast(idblock, s)
23 ∀w ∈ N(v), w ≠ s, edge_status_v[w],
24     send ⟨Deblock, idblock⟩ to w.
25 Cycle_Search(idblock);

26 Improve(y, deg, e, path)
27 send ⟨Remove, (v, y), deg, e, path⟩ to head(path)

28 Deblock(y, s)
29 if deg_v ≥ deg_y then Broadcast(v, s);
30 if deg_y ≥ deg_v then send ⟨Deblock, y⟩ to y

31 Reverse_Orientation((x, y), deg, (w, z), u, path)
32 let path = list_1 ⊕ v ⊕ list_2 and p=head(list_2).
33 if u = parent_v then
34     if ¬edge_status_v[p] then Reverse_Aux(p);
35     else
36         parent_v ← p; distance_v ← distance_p + 1;
37     send ⟨Remove, (x, y), deg, (w, z), path⟩ to p
38 else
39     let q=head(reverse(list_1)).
40     if ¬edge_status_v[q] then Reverse_Aux(q);
41     else
42         parent_v ← q; distance_v ← distance_q + 1;
43     send ⟨Back, (x, y),reverse(list_1)⟩ to q

44 Reverse_Aux(u)
45 send from u ⟨Reverse, v⟩ to parent_u, then wait
   and treat only InfoMsg_v until message
   ⟨Reverse, v⟩ was received.

46 target_remove(v, deg_max, (w, z)) ≡ (v = w ∨
   v = z) ∧ (deg_w = deg_max ∨ deg_z = deg_max)
   ∧ (edge_status_w[z] = true);

47 source_remove(v, (x, y)) ≡ (v = x ∨ v = y);
```

Figure 1: Algorithm's procedures at node v

Variables. This short section lists all the variables used by the algorithm. For any node $v \in V(G)$, $N(v)$ (we assume an underlying self-stabilizing protocol that regularly updates the neighbors set) denotes the set of all neighbors of v in the network G, and $\mathrm{ID}_v \in \mathbb{N}$ is the unique identifier of v. Nodes repeatedly send theirs variables to each of theirs neighbors u via a `InfoMsg` and update theirs local variables upon reception of this type of message from a neighbors. Each node v maintains the following variables:

- **Integer type variables:** $\underline{\mathrm{root}_v}$: the ID of the root of the spanning tree (computed by node v); $\underline{\mathrm{parent}_v}$: the parent ID of v in the spanning tree; $\underline{\mathrm{distance}_v}$: the distance of v to the root of the tree; $\underline{\mathrm{dmax}_v}$: Local estimation of k, it is updated upon the reception of a `InfoMsg` message. A change on k is detected via the $\mathrm{color\_tree}_v$ variable. This information is further disseminated in the network via `InfoMsg` messages; deg_v: the degree of v in the tree;
- **Boolean type variables:** $\underline{\mathrm{edge\_status}_v[u]}$: is true when $\{u, v\}$ is an edge of the tree; $\underline{\mathrm{color\_tree}_v}$: used to track any change on dmax_v;

Predicates. The predicates at node v are the following:

- $\mathrm{better\_parent}(v) \equiv \bigvee_{u \in N(v)} \mathrm{root}_v > \mathrm{root}_u$

```
1  Do forever: send InfoMsg_v to all u ∈ N(v)
2  Upon receipt of InfoMsg_u from u: Update_State(u, deg_u); Cycle_Search(NIL);
3  Upon receipt of ⟨Remove, (x, y), deg_max, (w, z), path⟩ ∧ locally_stabilized(v) from u:
4  if target_remove(v, deg_max, (w, z)) then
5      Reverse_Orientation((x, y), deg_max, (w, z), u, path); color_tree_v ← ¬color_tree_v;
6  else
7      if source_remove(v, (x, y)) then send ⟨UpdateDist, (w, z), distance_y + 1⟩ to parent_v;
8          parent_v ← y; distance_v ← distance_parent_v + 1;
9      else let path = list_1 ⊕ v ⊕ list_2 and p=head(list_2).
10         if w, z ∉ list_2 then
11             if ¬edge_status_v[u] then Reverse_Aux(head(list_2));
12             else parent_v ← head(list_2); distance_v ← distance_parent_v + 1;
13         send ⟨Remove, (x, y), deg_max, (w, z), path⟩ to head(list_2)
14 send InfoMsg_v to all w ∈ N(v)
15 Upon receipt of ⟨Back, (x, y), path⟩ ∧ locally_stabilized(v) from u:
16 if source_remove(v, (x, y)) then send ⟨UpdateDist, (w, z), distance_y + 1⟩ to parent_v;
17     parent_v ← y; distance_v ← distance_parent_v + 1;
18 else let path = list_1 ⊕ v ⊕ list_2 and p=head(list_2).
19     if ¬edge_status_v[u] then Reverse_Aux(p); else parent_v ← p; distance_v ← distance_p + 1;
20     send ⟨Back, (x, y), path⟩ to p
21 send InfoMsg_v to all w ∈ N(v)
22 Upon receipt of ⟨Deblock, idblock⟩ ∧ locally_stabilized(v) from u: Broadcast(idblock, u);
23 Upon receipt of ⟨Reverse, target⟩ ∧ locally_stabilized(v) from u:
24 if target ≠ v then send ⟨Reverse, target⟩ to parent_v; parent_v ← u;
25 Upon receipt of ⟨UpdateDist, (w, z), dist⟩ ∧ locally_stabilized(v):
26 if v = w ∨ v = z then distance_v ← dist + 1;
27     ∀u ∈ N(v), u ≠ parent_v ∧ edge_status_v[u], send ⟨UpdateDist, (w, z), distance_v⟩.
```

Figure 2: Algorithm at node v

- coherent_parent$(v) \equiv (\text{parent}_v \in N(v) \bigcup \{v\}) \wedge (\text{root}_v = \text{root}_{\text{parent}_v})$
- coherent_distance$(v) \equiv (\text{parent}_v \neq \text{ID}_v \wedge \text{distance}_v = \text{distance}_{\text{parent}_v} + 1) \vee (\text{parent}_v = \text{ID}_v \wedge \text{distance}_v = 0)$
- new_root_candidate$(v) \equiv \neg$coherent_parent$(v) \vee \neg$coherent_distance(v)
- is_tree_edge$(v, u) \equiv \text{parent}_v = \text{ID}_u \vee \text{parent}_u = \text{ID}_v$
- tree_stabilized$(v) \equiv \neg$better_parent$(v) \wedge \neg$new_root_candidate(v)
- degree_stabilized$(v) \equiv \bigwedge_{u \in N(v)} \text{dmax}_v = \text{dmax}_u$
- color_stabilized$(v) \equiv \bigwedge_{u \in N(v)}$ color_tree$_v =$ color_tree$_u$
- locally_stabilized$(v) \equiv$ tree_stabilized$(v) \wedge$ color_stabilized(v)

Procedures. The procedures at node v are the following:

- change_parent_to$(v, u) \equiv \text{root}_v \leftarrow \text{root}_u; \text{parent}_v \leftarrow \text{ID}_u; \text{distance}_v \leftarrow \text{distance}_u + 1$
- create_new_root$(v) \equiv \text{root}_v \leftarrow \text{ID}_v; \text{parent}_v \leftarrow \text{ID}_v; \text{distance}_v \leftarrow 0$

Messages. The messages used by our algorithm are the following:

- $\langle$`InfoMsg`, root_v, parent_v, distance_v, dmax_v, deg_v, $\text{edge\_status}_v[u]$, $\text{color\_tree}_v\rangle$. This message is used to gossip the local variables of a node.
- $\langle$`Search`, *init_edge*, *idblock*, *path*$\rangle$: it is used to find the fundamental cycle induced by a non tree edge, where *init_edge* is the initiate non tree edge, *idblock* is the ID of a blocking node and *path* is the fundamental cycle.
- $\langle$`Remove`, *init_edge*, *deg_max*, *target*, *path*$\rangle$: it is used to reduce the degree of a maximum degree node by deleting an adjacent edge then it changes edge orientation in the fundamental cycle, where *init_edge* is the initiate non tree edge, *target* is the edge to be deleted, *deg_max* is the maximum degree of *target* extremities and *path* is the fundamental cycle.
- $\langle$`Back`, *init_edge*, *path*$\rangle$: change edge orientation in the fundamental cycle after an improvement, where *init_edge* is the initiate non tree edge and *path* is the fundamental cycle.
- $\langle$`Deblock`, *idblock*$\rangle$: it is used to change the state of a blocking node, where *idblock* is the ID of a blocking node.
- $\langle$`Reverse`, *target*$\rangle$: it is used to change the edge orientation in the fundamental cycle where *target* is the ID of the node when the change orientation stops.
- $\langle$`UpdateDist`, *target*, *dist*, *path*$\rangle$: update the distance of fundamental cycle nodes, where *target* is the edge when the update stops, *dist* is the distance from the tree root of the sender and *path* is the fundamental cycle.

Note that the *path* information is never stored at a node, therefore it is not listed as local variable of a particular node. However this information is carried by different messages. Therefore the complexity of our solution in the length of the network buffers is at least $O(n \log n)$ (the maximal length of the *path* chain) as explained in the complexity section.

3.2 Elementary building blocks

In this section, we provide first a detailed description of the underlying modules for our degree reduction algorithm, namely: a spanning tree module, a module that detects fundamental cycles and finally a module that computes and disseminates the maximal degree of the spanning tree. We conclude the section by the detailed presentation of the degree reduction module.

3.2.1 Spanning tree module

The algorithm described below is a simplification of the BFS algorithm proposed in [1]. Each node v maintains three variables: the local known ID of the tree root, a pointer to v's parent and the distance of v to the spanning tree root. The output tree is rooted at the node having the minimum root value.

The algorithm uses two rules formally specified below. The first rule, "correction parent", enables the update of the locally known root: for a node v if a neighbor u has a lower root ID than v then v changes its root to u's root, and u becomes the parent of v. If several neighbors verify this property then v will choose as parent the node with the minimal ID. This choice is realized by the argmin operation. The second rule, "correction root", creates a root if the neighborhood of a node has an incoherent state. In a coherent state, the distance

in the BFS tree with respect to the root should be the distance of the parent plus 1. In the case of the root this distance should be 0. Additionally, the parent of a node should be a neighbor.

R1: correction parent: If $\neg$new_root_candidate(v) $\wedge$ better_parent(v)
then change_parent_to(v,argmin{root$_u$: $u \in N(v)$});

R2: correction root:. If new_root_candidate(v) then create_new_root(v);

3.2.2 Fundamental cycle detection module

We recall that given G a network, T a spanning tree of G and $e = \{u, v\}$ an edge of G that is not in T, the cycle C_e generated by adding e to T is called *fundamental*.

Computing a fundamental cycle at an arbitrary node v is performed by Procedure **Cycle_Search**(v) (see Figure 3). For each non tree edge $\{u, v\}$ such that $\mathrm{ID}_v < \mathrm{ID}_u$ (v is the initiator for the non tree edge $\{u, v\}$), node v initiates a DFS in the current spanning tree T to discover the path in the tree between u and v (Figure 3, lines 1-3). This DFS uses messages of type `Search`. The DFS propagation (Figure 3, lines 4-6) discovers only the tree edges in the fundamental cycle of $\{u, v\}$. It updates *path* by appending the discovered node and its degree in the tree.

The DFS search propagation stops (Figure 3, lines 7-8) when the other extremity of the edge (i.e. u) is reached and *path* contains only the tree edges in the fundamental cycle of $\{u, v\}$.

```
1 Cycle_Search(id)
2 if locally_stabilized(v) then
3     ∀u ∈ N(v) with edge_status_v[u] = false and ID_v < ID_u: initiate a DFS propagation of a Search
      message crossing only tree edges by sending ⟨Search, (u, v), id, [{v, deg_v}]⟩ to w ∈ N(v) s.t.
      edge_status_v[w] = true

4 Upon receipt ⟨Search, (x, y), id, path⟩ ∧ locally_stabilized(v) from u:
5 if v ≠ x then
6     DFS propagation on tree edges of ⟨Search, (x, y), id, path ⊕ [{v, deg_v}]⟩
7 else
8     if ¬edge_status_v[y] then Action_on_Cycle(id, y, path, u);
```

Figure 3: Fundamental cycle detection at node v

3.2.3 Maximum degree module

Our construction of minimum-degree spanning tree from an arbitrary spanning tree requires at every node v the knowledge of: (1) which of its adjacent edges belongs to the spanning tree; (2) the maximum degree of the current spanning tree.

Note that both information can be computed using only local communication as it will be described in the sequel. The coherent maintenance of the edges part of the spanning tree is realized via the variable edge_status$_v[u]$, updated by `InfoMsg` messages. This process is similar with the maintenance of dynamic trees [14]. The computation of the maximum

node-degree of the current spanning tree, can be done using a Propagation of Information with Feedback (PIF) protocol [16, 17]. That is, the protocol selects in the feedback phase at each level from the leaves to the root the maximum node-degree (by using deg_v). In the propagation phase that can be piggybacked on the `InfoMsg` messages the root disseminates the new maximum degree to all the nodes in the spanning tree. Upon the reception of `InfoMsg` a node updates its $dmax_v$ variable.

Note that the maximal degree of the tree changes during the execution of the algorithm via the degree reduction procedure. When a node of maximum degree has its degree decreased by one, the algorithm uses color_tree$_v$ variable to detect this change (see line 5, Figure 2). Additionally, a node that detects an incoherence in its neighborhood related to the maximum degree blocks the construction of the minimum-degree spanning tree until the neighborhood becomes locally_stabilized (all neighbors have the same value for their color_tree variable).

3.2.4 Degree reduction module

For all edges that do not belong to the current tree T our algorithm proceeds as follows. For each of these edges e, one of its adjacent nodes computes the fundamental cycle C_e, and checks whether e is an improving edge, and whether the degree of C_e is equal to $\deg(T)$. If the two latter conditions are satisfied, then e is swapped with one edge of C_e incident to a node of maximum degree in T. This operation is repeated until there is no more improving edge e with degree of C_e equal to the degree of the current tree. If a blocking node for C_e is encountered, then the algorithm tries to reduce the degree of the blocking node, recursively, in a way similar to the one used for decreasing the degree of the nodes of maximum degree. When no more improvement is possible then the algorithm stops. Note that [9] proved that the degree of a tree for which no improvement is possible is at most $\Delta^* + 1$, where Δ^* is the degree of a MDST of G. In the following we describe in details the above reduction process.

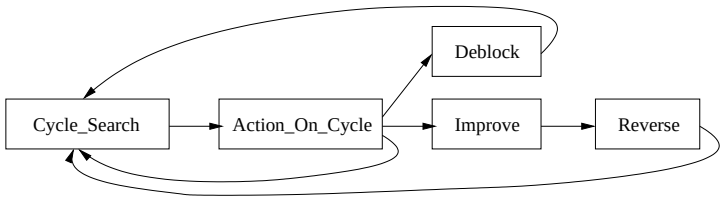

Figure 4: Degree reduction process

Cycle_Search: Repeatedly each non tree edge looks for its fundamental cycle by using the procedure **Cycle_Search** is described in Section 3.2. When a non tree edge $e = \{u, v\}$ discovers its fundamental cycle (Figure 3, line 8) it checks whether it is an improving edge in procedure **Action_on_Cycle**.

Action_on_Cycle: If the fundamental cycle contains no maximum degree node, no improvement is possible for this fundamental cycle. Otherwise (line 8) if the fundamental cycle also contains no blocking node (line 6) an improvement is possible (procedure **Improve** is called, lines 11-14). If there are blocking nodes the procedure **Deblock** is started (lines 9-

10). The procedure tries to improve the blocking node (lines 18-21), either via the procedure **Improve** or via the procedure **Deblock** (lines 15-17).

Improve: The procedure **Improve** sends a `Remove` message (lines 26-27), which is propagated along the fundamental cycle (Figure 2, lines 9-13). When the `Remove` message reaches its destination (edge $e' = \{w, z\}$), the degree and the status of e' are checked. If the maximum degree or the edge status have changed, then the `Remove` message is discarded. Otherwise, the status of edge e' is modified (it becomes a non-tree edge), and a message is sent along the cycle for correcting the parent orientation (procedure **Reverse_Orientation** explained below). When `Remove` reaches the edge e that initiated the process then e is added to the tree (Figure 2, lines 7-8). In the case when `Remove` message meets a deleted edge on its path, it carries on as if the deleted edge would be still alive.

Reverse_Orientation: After the remove of an edge e', the orientation of the fundamental cycle must be corrected (see Figure 5). This is achieved via the circulation of two messages: `Back` and `Remove`. The procedure **Reverse_Orientation** deletes e' and checks following the orientation of e' whether a `Back` or a `Remove` message must be used to correct the orientation. If e' is oriented opposite to the direction followed by `Remove` message, then a `Remove` message is used (Figure 5(a)) otherwise a `Back` message is used (Figure 5(b)). The result of **Reverse_Orientation** on the path from e' to e on the fundamental cycle of e can be seen in Figure 5(c).

Note that a `Remove` (or `Back`) message propagated along the fundamental cycle may meet an edge deleted by another improvement (see message `Reverse` Figure 5). This implies that the fundamental cycle of e has changed. In this case it is necessary to finish the improvement operation otherwise the tree partitions.

Deblock: To perform an edge exchange with a blocking node $w \in \{u, v\}$, our algorithm starts by reducing by one the degree of w. For this purpose, w broadcasts a `Deblock` message in all its sub-tree (see procedure **Deblock**). When a descendant w' of w, adjacent to a non tree edge f, receives a `Deblock` message, it proceeds with the discovery of the fundamental cycle C_f of f as described in Section 3.2. If this fundamental cycle C_f enables to decrease the degree of the blocking node w, then w is not anymore blocking, and the swap e with e' can be performed. However, if the fundamental cycle C_f does not enable to decrease the degree of the blocking node w, then the procedure carries on recursively by broadcasting a `Deblock` messages for w'.

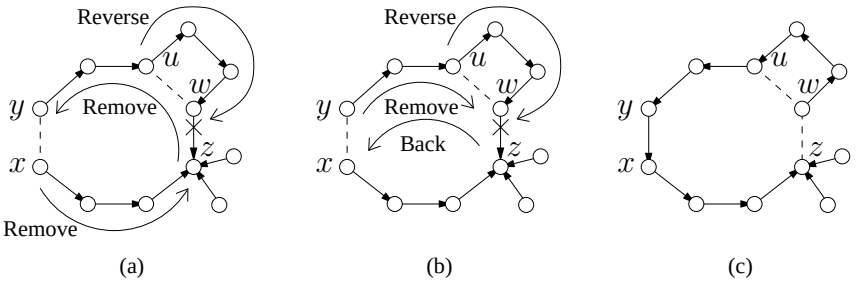

Figure 5: Illustration of Reverse_Orientation and Reverse_Aux procedures: (a) when $\mathrm{ID}_x > \mathrm{ID}_y$, (b) when $\mathrm{ID}_x < \mathrm{ID}_y$, (c) the resulted network given by (a) and (b).

Note that in order to maintain the tree stabilized we must update the distance of nodes on the path reversed after the remove of e', as their distances to the tree root has changed. Therefore an `UpdateDist` message is diffused for all children on the reversed path.

4 Algorithm correctness

In the sequel we address different aspects related to the corruption of the local state of a node.

`InfoMsg` messages ensure the coherence between the copy of a node variables and their original values. Therefore, even if a node maintains corrupted copies their values will be corrected as soon as the node receives `InfoMsg` messages from each of its neighbors.

The correctness of the computation and the dissemination of the maximum degree of the current tree follow directly from the correctness of the self-stabilizing PIF scheme [17].

The reduction procedure and the search of fundamental cycles are frozen at a node v until the neighborhood of v is locally stabilized. That is, modules executed at an arbitrary node are locally ordered on priority bases: the module with maximal priority being the spanning tree construction, followed by the degree reduction module.

In the following we prove that the execution of the reduction procedure is blocked for a finite time (i.e. the time needed to the algorithm to compute a spanning tree). That is, we prove that the algorithm computes in a self-stabilizing manner a spanning tree in a finite number of steps.

Lemma 1 *Starting from an arbitrary configuration, eventually all nodes share the same root value.*

Proof. Nodes periodically exchange `InfoMsg` messages with their neighborhood. Therefore, even if the local copy of the neighbors variables are not identical with the original ones, upon the reception of a `InfoMsg` message a node corrects its local copies (via the procedure **Update_State**). Assume w.r.g. two different root values coexist in the network. Let $v_1 < v_2$ be these values. Let k be the number of nodes having their root variable setted to v_2. Since the network is connected there are two neighboring nodes p_1 and p_2 such that the local root of p_1 is v_1 and the local root of p_2 is v_2. After the reception of an `InfoMsg` from p_1, p_2 detects its inconsistency and changes the local root value to v_1 and the parent variable to $p1$ by executing the rule "correction parent". Hence the number of root variables setted to v_2 decreases by 1. Recursively, the number of root variables setted to v_2 falls to 0.

Note that the root values are not modified by the other procedures of the algorithm. Overall, all nodes in the network will have the same root value. □

Lemma 2 *Starting from an arbitrary configuration, eventually an unique spanning tree is alive in the network.*

Proof. Assume the contrary: two or more spanning trees are created. This leads to either the existence of two or more different root values or the presence of at least one cycle. Since

the values of the root variable are totally ordered then by Lemma 1 one of those values will eventually won the contest which invalidates the multiple roots assumption.

Assume there is cycle in the network. Note that there is an unique root and each node has a distance to the root which is the distance of the parent plus one. In a cycle there is at least one node such that the coherent_distance predicate returns false. This node will eventually execute the "correction root" rule. According to the proof of Lemma 1 the new root will run a competition with the existing root and the root with the lowest ID remains root while the other one modifies its root and distance variables. □

Note that the minimal degree reduction procedure interfere with the tree maintenance via the modification of the distance variable in the edge reversal process. The messages of type `UpdateDist` are in charge of correcting the distance value and the parent value. During the correction of a path in the tree the messages related to the tree reduction are frozen until the correction is completed. Therefore, after the edge reversal completes the tree is coherent.

Now we prove that our algorithm maintains the same performance as algorithm in [9] and converges to a legitimate configuration verifying the hypothesis of Theorem 1 ([9]).

Theorem 1 (Fürer and Raghavachari [9]) *Let T be a spanning tree of degree Δ of a graph G. Let Δ^* be the degree of a minimum-degree spanning tree. Let S be the set of vertexes of degree Δ in T. Let B be an arbitrary subset of vertexes of degree $\Delta - 1$ in T. Let $S \cup B$ be removed from the graph, breaking the tree T into a forest F. Suppose G satisfies the condition that there are no edges between different trees in F. Then $\Delta \leq \Delta^* + 1$.*

We call *improvement* a step of our algorithm in which the degree of at least one node of maximum degree is decreased by 1, without increasing the degree of a node which already has the maximum degree, nor the degree of a blocking node.

Lemma 3 *When applying procedure* **Improve***, our algorithm makes an improvement.*

Proof. Let $e = \{u, v\}$ be an improving edge for the current tree T of maximum degree k. Before adding e to T in the procedure **Improve**, some extremity of the edge e sends a `Remove` message along C_e to delete some edge $e' = \{x, y\}$ of the cycle, adjacent to a node of degree k or $k-1$. This message `Remove` is routed along C_e. When it reaches one extremity of edge e', here are two cases:

Case 1: e' is still a tree edge of degree k or $k-1$. Then, according to procedure **Improve**, edge e' is removed from T, and u or v is informed by a `Back` message that e can be added to the tree T. In this case, we have an improvement.

Case 2: e' is no more a tree edge of degree k or $k-1$. This implies that the degree of C_e has been decreased (by a concurrent improving edge), yielding an improvement. In this case, according to procedure **Improve**, the message `Remove` is discarded, for preserving the spanning tree property. In both cases, an improvement occurred. □

A blocking node w in the current tree T is said *eventually non blocking* if it is non blocking, or there is an improving edge $e = \{u, v\}$ connecting two nodes u and v in the sub-tree T_w of T rooted at w, such that $w \in C_e$, and u and v are eventually non blocking.

Lemma 4 *Let w be a blocking node which can be made non blocking. Then procedure* **Deblock**(w) *reduces the degree of w by 1, without increasing the degree of neither a node with maximum degree, nor a blocking node.*

Proof. The proof proceeds by induction on the number of calls to procedure **Deblock**. If at the first call to **Deblock** there is an improving edge in the sub-tree of w (according to [9] it is sufficient to look for improving edge in the sub-tree of w) then an improvement is performed by the procedure **Improve** reducing the degree of w according to Lemma 3. If there is no improving edge but at least one blocking node adjacent to a non tree edge in the sub-tree of w which contains w in its fundamental cycle, then the procedure **Deblock** is recursively called by these ones according to procedure **Deblock_Node**. So if an improvement is possible, this improvement is propagated to w by a sequence of improvements. Note that each improvement can only increase the degree of nodes with a degree $\leq k-2$ according to Lemma 3, so procedure **Deblock**(w) can not increase the degree of neither a node with maximum degree, nor a blocking node. □

Theorem 2 *The algorithm returns a spanning tree of G with degree at most $\Delta^* + 1$.*

Proof. In the following we prove that a legitimate configuration verifies the hypothesis of Theorem 1. That is, when our algorithm reaches a legitimate configuration, there is no more possible improvement. Assume the contrary. We assume there is a tree structure T with maximum degree k. Suppose there is a possible improvement for T, by performing the swap between the edges $e_1 \notin T$ and $e_2 \in T$ to obtain the tree $T' = T \cup \{e_1\} \backslash \{e_2\}$ and assume algorithm does not perform this improvement. This implies that edge e_1 has a degree equal to k, otherwise as showed by Lemmas 3 and 4 the node of minimum id of e_1 will perform an improvement with procedure **Improve** or run **Deblock** to reduce the degree of e_1, according to procedure **Action_on_Cycle**. This contradicts the fact that e_1 can be used to perform an improvement, and so if there is an improvement for T then the algorithm will perform this improvement. □

5 Complexity issues and Discussions

The following lemma states that the time complexity is polynomial while the memory complexity is logarithmic in the size of the network.

Lemma 5 *Our algorithm converges to a legitimate state for the MDST problem in $O(mn^2 \log n)$ rounds using $O(\delta \log n)$ bits memory in the send/receive model[3], where δ is the maximal degree of the network.*

Proof. The algorithm is the composition of three layers: the spanning tree construction, the maximum degree computation and the tree degree reduction layer. The first and the second layer have respectively a time complexity of $O(n^2)$ rounds [1, 11] and $O(n)$ rounds [16]. The time complexity is dominated by the last layer. According to [9], there are $O(n \log n)$

[3]In the classical message passing model the memory complexity is $O(\log n)$

phases. A phase consists of decreasing by one the maximum degree in the tree (except for the last phase). In fact, let k the degree of the initial tree, there are $O(n/k)$ maximum degree nodes. As $2 \leq k \leq n-1$, summing up the harmonic series corresponding to the k values leads to $O(n \log n)$ phases. During each phase a non tree edge $\{u, v\}$ performs the following operations: finding its fundamental cycle and achieving an improvement (if it is an improving edge). To find its fundamental cycle, $\{u, v\}$ uses the procedure **Cycle_Search** which propagates a `Search` message via a DFS traversal of the current tree. So the first operation requires at most $O(n)$ rounds. For the second operation there are two possible cases: a direct and an indirect improvement. A direct improvement is performed if $\{u, v\}$ is an improving edge, in this case a `Remove` message is sent and follows the fundamental cycle to make the different changes, requiring $O(n)$ rounds. An indirect improvement is performed when u or v are blocking node ($\{u, v\}$ is a blocking edge). In the worst case, the blocking edges chain is of size at most $m-(n-1)$. Therefore $m-(n-1)$ exchanges are necessary to reduce the tree degree. This requires $O(mn)$ rounds (according to the direct improvement). So as the second operation needs at most $O(mn)$ rounds, the third layer requires $O(mn^2 \log n)$ rounds.

In the following we analyze the memory complexity of our solution. Each node maintains a constant number of local variables of size $O(\log n)$ bits. However, due to specificity of our model (the send/receive model) the memory complexity including the copies of the local neighborhood is $O(\delta \log n)$ where δ is the maximal degree of the network. □

Note that the messages exchanged during the execution of our algorithm carry information related to the size of fundamental cycles. Therefore, the buffers length complexity of our solution is $O(n \log n)$.

6 Conclusion and open problems

We proposed a self-stabilizing algorithm for constructing a minimum-degree spanning tree in undirected networks. Starting from an arbitrary state, our algorithm is guaranteed to converge to a legitimate state describing a spanning tree whose maximum node degree is at most $\Delta^* + 1$, where Δ^* is the minimum possible maximum degree of a spanning tree of the network.

To the best of our knowledge our algorithm is the first self-stabilizing solution for the construction of a minimum-degree spanning tree in undirected graphs. The algorithm can be easily adapted to large scale systems since it uses only local communications and no node centralizes the computation of the MDST. That is, each node exchanges messages with its neighbors at one hop distance and the computations are totally distributed. Additionally the algorithm is designed to work in any asynchronous message passing network with reliable FIFO channels. The time complexity of our solution is $O(mn^2 \log n)$ where n and m are respectively the number of nodes and edges in the network. The memory complexity is $O(\delta \log n)$ in the send/receive atomicity model.

Several problems remain opened. First, the computation of a minimum-degree spanning tree in directed topologies seems to be a promising research direction with a broad area

of application (i.e. sensor networks, adhoc network, robot networks). Second, the new emergent networks need scalable solutions able to cope with nodes churn. An appealing research direction would be to design a super-stabilizing algorithm [18] for approximating a MDST.

References

[1] Y. Afek and S. Kutten and M. Yung. Memory-efficient self stabilizing protocols for general networks. *4th International Workshop on Distributed Algorithm, WDAG, Springer LNCS volume 486*, 15-28, 1991.

[2] A. Arora and M.G. Gouda. Distributed Reset. *In Proceedings of the Tenth Conference on Foundations of Software Technology and theoretical Computer Science, FSTTCS*, 17-19, 1990.

[3] L. Blin. and F. Butelle. The first approximated distributed algorithm for the minimum degree spanning tree problem on general graphs. *Int. J. Found. Comput. Sci.*, 15(3):507-516, 2004.

[4] F. Butelle and C. Lavault and M .Bui. A Uniform Self-Stabilizing Minimum Diameter Tree Algorithm (Extended Abstract). *Distributed Algorithms, 9th International Workshop (WDAG)*, 257-272, 1995.

[5] J. Burman and S. Kutten. Time Optimal Asynchronous Self-stabilizing Spanning Tree. *Distributed Computing, 21st International Symposium (DISC)* 92-107, 2007.

[6] E.W. Dijkstra. Self-stabilizing systems in spite of Distributed Control. *Communications of the ACM*, 17(11):643-644, 1974.

[7] S. Dolev and A. Israeli and S. Moran. Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity. *in ninth annual ACM symposium on Principles of distributed computing (PODC)*, 103-117, 1990.

[8] M. Fürer and B. Raghavachari. Approximating the minimum degree spanning tree to within one from the optimal degree. In *the Proc. of the 3rd ACM-SIAM Symp. on Discr. Algo. (SODA)*, 317-324, 1992.

[9] M. Fürer and B. Raghavachari. Approximating the minimum degree steiner tree to within one of optimal. *Journal of Algorithms*, 17:409-423, 1994.

[10] R.G. Gallager and P.A. Humblet and P.M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66-77, 1983.

[11] F.C. Gärtner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. *TR,EPFL*, October 2003.

[12] L. Higham and Z. Liang. Self-Stabilizing Minimum Spanning Tree Construction on Message-Passing Networks. *Distributed Computing, 15th International Conference (DISC)*, 194-208, 2001.

[13] T. Hérault and Pierre Lemarinier and Olivier Peres and Laurence Pilard and Joffroy Beauquier. A Model for Large Scale Self-Stabilization. IPDPS 2007.

[14] Silvia Bianchi and Ajoy Kumar Datta and Pascal Felber and Maria Gradinariu. Stabilizing Peer-to-Peer Spatial Filters. ICDCS 2007.

[15] Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network RESET (extended abstract). PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing.

[16] Lélia Blin and Alain Cournier and Vincent Villain. An Improved Snap-Stabilizing PIF Algorithm. In 6th Symposium on Self-Stabilizing Systems (SSS), LNCS 2704, pages 199-214, 2003.

[17] Alain Cournier and Ajoy Kumar Datta and Franck Petit and Vincent Villain. Optimal snap-stabilizing PIF algorithms in un-oriented trees. J. High Speed Networks, volume 14, number 2, pages 185-200, 2005.

[18] Shlomi Dolev. Self-Stabilization. MIT Press, 2000.